6. Транслатори

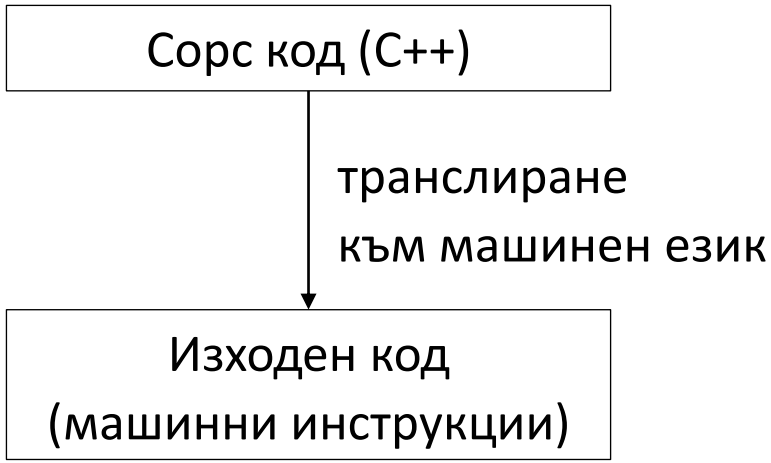
Проф. д-р Емил Хаджиколев

1. Транслатори - компилатори и интерпретатори;
2. Класификация на езиците за програмиране според начина за транслиране;
3. Структура на C++ програма;
4. Компилиране на C++ програма.

Транслатор (1)

- translator – преводач.
- Транслаторът е специална програма.
- При компютърните езици, транслаторът обикновено превежда сорс код, написан на език на високо ниво, към машинен език.
  - Може превода да не е към машинен език.
- Една програма написана на език от високо ниво може да бъде транслирана за различни машини чрез специфични транслатори.
  - всяка конструкция на език от високо ниво се превежда автоматично от транслатора до няколко съответни машинни инструкции (и данни към тях).

Транслатор (2)

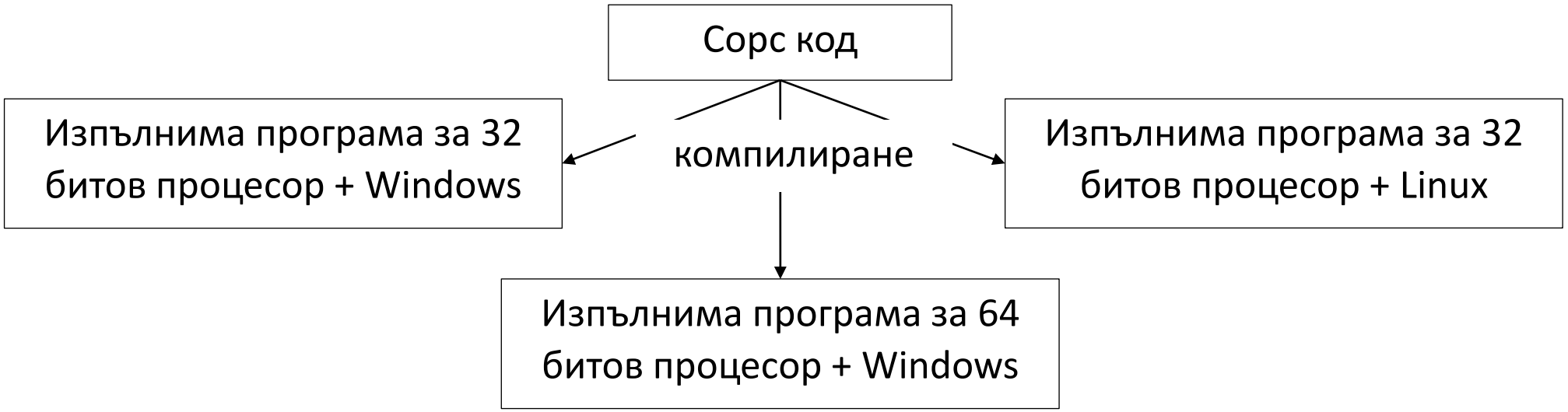


- Транслаторът **превежда програми, написани на един ЕП** (наречен входен) **в програми на друг език** (изходен).
  - Входният код (напр. на C++) наричаме сорс код (source code).
- Не е задължително изходните програми да съдържат инструкции за физическа машина.
- Възможно е изходните програми да съдържат инструкции към виртуална машина (специална програма – абстрактен процесор). От своя страна, виртуалната машина превежда дадените инструкции до машинен език.

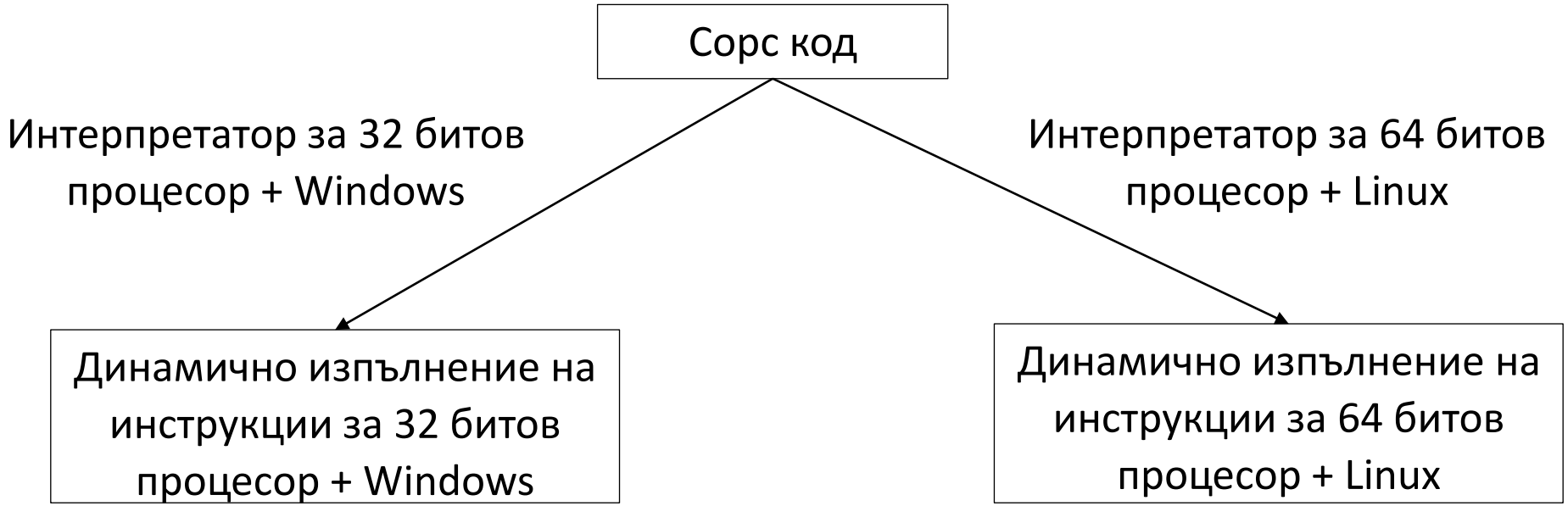
Компилатори и интерпретатори

- Има **два вида транслатори**:
  - Компилатор;
  - Интерпретатор.
- **Компилаторът създава** еднократно **изпълнима програма** (с разширение exe в Windows), която може да бъде изпълнявана независимо от сорс кода на входната програма.
- **Интерпретаторът чете сорс кода и го превежда на части като не създава изпълнима програма**. Интерпретаторът се стартира при всяко изпълнение на програмата.

Примерни компилации



Примерни интерпретации



Класификация на езиците за програмиране според начина за транслиране

- **Компилативни** (компилаторни) **езици** (Compiled Languages) – C, C++,Go, Rust и др.
- **Интерпретативни** (интерпретаторни) **езици** (Interpreted Languages). Първоначално скриптовите езици са били само интерпретативни – PHP, JavaScript и др. – но почти не са останали чисто интерпретативни езици.
- **Хибридни** – комбинират и двата подхода (Java, C#, PHP, Python, JavaScript и др.). Java прави хибридния модел популярен:
  - компилаторът на Java създава изпълним код (наречен байт код) за виртуалната машина (ВМ) на Java – файлове с разширение class.
  - ВМ интерпретира изпълнимите class-файлове и динамично създава машинни инструкции за конкретната машина.
  - Предимства: Компилираният байт кодът е сигурен и без грешки, малък като обем, лесно преносим/изпълним на всяка платформа с ВМ на Java.

Предимства и недостатъци на подходите за транслиране (1)

- Ако дадена програма е компилирана успешно, то в нея със сигурност няма синтактични грешки (но може да има логически грешки).
- При интерпретацията, синтактичните грешки в сорс кода (обикновено) се откриват по време на изпълнение на кода.
  - Поради това синтактични грешки може да бъдат прихванати на много по-късен етап и то не от програмиста, а от потребител. (Правилното тестване би трябвало да се справя с подобни проблеми.)
  - Някои интерпретатори изпълняват предварителна обработка за синтактични грешки.
- Изпълнимите програми се изпълняват по-бързо от тези, които се интерпретират.
- При компилативните езици, всяка промяна на сорс кода изисква ново компилиране и се създава отново изпълним файл, а при скриптовите – няма създаване на допълнителни (изпълними) файлове.

Предимства и недостатъци на подходите за транслиране (2)

- Има компилатори и интерпретатори за различни машини (машина = процесор + операционна система).
  - Изпълнима програма се създава за конкретна машина чрез конкретен съответен компилатор. Някои компилатори може да компилират за различни машини (като им се указват параметри).
  - При интерпретаторите, аналогично, има различни интерпретатори за различни машини, които интерпретират сорс кода по-подходящ за съответната машина начин.
- При компилаторите, сорс кода е скрит от крайния потребител. (Има декомпилатори, с които може да се види началния код в някакъв вид – не с имената зададени от програмиста; синтактичните конструкции може да са променени и др.)
- При интерпретаторите, сорс кода се разпространява в явен вид. Крайният потребител не е задължително да може да види сорс кода - javaScript се вижда в уеб страниците, но php и др. подобни кодове за създаване на уеб страниците – не.

Структура на C++ програма

- Програмата на C++ се описва в текстов файл с разширение cpp.
- В кода на програмата се:
  - включват стандартни библиотеки: с директива #include, последвана от името на библиотеката във ъглови скоби <>:

    ```
    #include <iostream>
    ```

  - Включват заглавни файлове, създадени от програмиста: отново с #include, но името на файла се задава в кавички:

    ```
    #include "myfile.h"
    ```

    Указаният файл се търси в текущата директория (може да се опише и път). В заглавните файлове може да се описват декларации на променливи или функции, които се дефинират в текущия „cpp“-файл.
  - Указват използвани области на имената:

    ```
    using namespace std;
    ```

  - Дефинират глобални променливи (и константи) и функции
  - Описва основната функция от която стартира програмата: `int main()`

Области на имената

- Една област на имената **namespace групира** различни обекти: **функции, променливи и др. под общо име**.
- Namespace-ът **задава контекст, в който дефинираме имена за различни елементи**.
- Възможно е да имаме функции и променливи с едни и същи декларации, но в различни namespace-ове.
- **Квалифицираното (пълното) име** на елемент от дадена област е:

  <име_на_област>::<име_на_елемент>
- **std е стандартна област на имената,** в която са дефинирани множество елементи.
- Може да използваме пълните имена или кратките, но само ако преди това укажем

  using namespace <име_на_област>;

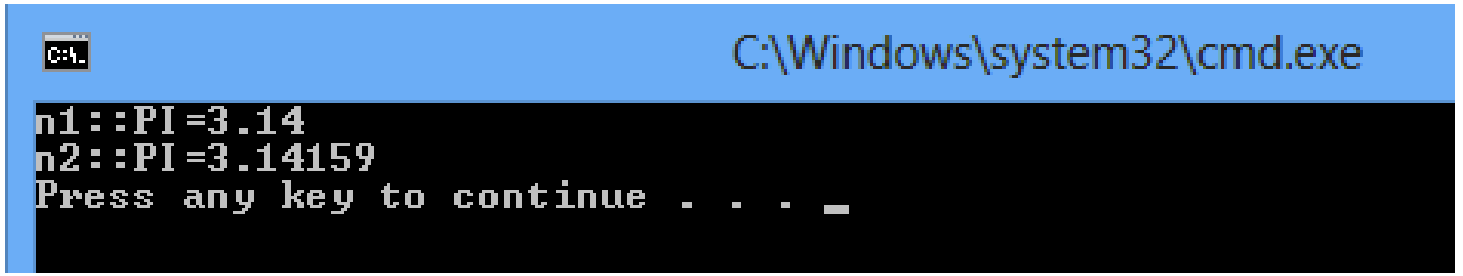
Дефиниране и използване на области на имената

```
#include <iostream>
namespace n1 { // създаване на namespace с име n1
    const double PI=3.14;
    void f() {  // пълното име на cout е std::cout
        std::cout << "n1::PI=" << PI << "\n";
    }
}
namespace n2 {// създаване на namespace с име n2
    const double PI = 3.14159;
    void f() {
        std::cout << "n2::PI=" << PI << "\n";
    }
}
```

```
int main(){
    // извикване на функцията f от n1
    n1::f();

    // извикване на функцията f от n2
    n2::f();

    return 0;
}
```



Директиви към предпроцесора

- **Предпроцесорът** е програма, която **се стартира преди** същинския процес на **компилиране**.
- **Предпроцесорът модифицира сорс кода на програмата с указани от директивите действия**.
- Има различни директиви:
  - включване на библиотеки (#include)
  - за условни включвания на код (#if, #ifdef, #ifndef, #elif, #else, #endif);
  - дефиниране на макроси (#define, #undef);
  - и др. (#pragma, #error);
- **Директивите не са част от езика C++.**

Процес на транслиране

При процеса на транслиране, файлът с изходния (сорс) код се обработва няколко пъти (на няколко паса-pass). Основните етапи са:

- **Стартиране на предпроцесор**, който преобразува изходния код (според предписанията на директивите) и създава „чист“ C++ код;
- **Компилиране** – създава т.нар. обектен код на програмата;
- **Стартиране на свързващ редактор** – свързва обектния код с указаните в програмата библиотеки и създава изпълнима програма.

Фази при транслиране (1)

По-точно, стандартният процес на транслиране се разделя на 9 фази (при някои компилатори може да са по-малко), които няма да разглеждаме подробно (http://en.cppreference.com/w/cpp/language/translation_phases):

- Преди стартиране на предпроцесора: символите на кода се преобразуват до основна форма (ASCII символи), като символите от други кодови таблици се преобразуват до ескейп-последователности (описана с ASCII символи); премахват се излишни празни символи; изходният код се декомпозира на token-и: коментари (които се премахват), заглавни библиотеки, идентификатори, символни (char) и низови литерали, числови литерали; оператори и други специални символи;
- Предпроцесорът изпълнява директивите, при което за всеки от включените с include файлове се стартира рекурсивно текущата процедура;

Фази при транслиране (2)

- След предпроцесора символните и низови литералите се преобразуват в подходящ за показване вид (от ескейп последователности до UTF-8 или друга кодировка);
- При компилирането се извършва синтактичен и семантичен анализ и token-ите се преобразуват в части (units) за транслация. Всяка такава част се асоциира със стандартен шаблон за транслация, който се инстанциира (определят му се конкретни параметри) и се съставя инстанциирана част (instantiation unit);
- Всички части, библиотеки, ресурси се свързват в програма.